TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Optimalizace datových struktur výpočetních programů s ohledem na využití cache

Bakalářská práce

Studijní program: B2646 – Informační technologie
Studijní obor: 1802R007 – Informační technologie

Autor práce: **Petr Král**
Vedoucí práce: Mgr. Jan Březina, Ph.D.

Liberec 2018

TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies

Cache optimization of data structures in numerical software

Bachelor thesis

Study programme: B2646 – Information technology
Study branch: 1802R007 – Information technology

Author: **Petr Král**
Supervisor: Mgr. Jan Březina, Ph.D.

Liberec 2018

Technická univerzita v Liberci
Fakulta mechatroniky, informatiky a mezioborových studií
Akademický rok: **2017/2018**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Petr Král**
Osobní číslo:	**M15000029**
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Optimalizace datových struktur výpočetních programů s ohledem na využití cache**
Zadávající katedra:	**Ústav nových technologií a aplikované informatiky**

Zásady pro vypracování:

1. Seznamte se s výpočetním software Flow123d.
2. Použijte výrazové šablony a paměťové optimalizace na výpočty používající knihovnu Armadillo.
3. Monitorujte využití cache v algoritmech assemblace matic v metodě konečných prvků pomocí nástroje cachegrind.
4. Otestujte a aplikujte vybrané optimalizace v datových strukturách pro uložení výpočetní sítě vzhledem k využití cache pamětí.
5. Demonstrujte zrychlení algoritmů asemblace lineárních systémů pomocí upravených datových struktur.

Rozsah grafických prací: **dle potřeby**

Rozsah pracovní zprávy: **30 - 40 stran**

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

[1] Kowarschik M., Wei C. (2003) An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In: Meyer U., Sanders P., Sibeyn J. (eds) Algorithms for Memory Hierarchies. Lecture Notes in Computer Science, vol 2625. Springer, Berlin, Heidelberg
[2] Cachegrind: a Cache-miss Profiler. [online]. [cit. 1.10. 2017] Dostupné na: http://valgrind.org/docs/manual/cg-manual.html
[3] J. Levon, OProfile Manual, http://oprofile.sourceforge.net/doc/, Victoria University of Manchester.

Vedoucí bakalářské práce: **Mgr. Jan Březina, Ph.D.**
Ústav nových technologií a aplikované informatiky

Datum zadání bakalářské práce: **19. října 2017**

Termín odevzdání bakalářské práce: **14. května 2018**

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

L.S.

Ing. Josef Novák, Ph.D.
vedoucí ústavu

V Liberci dne 19. října 2017

Prohlášení

Byl jsem seznámen s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu TUL.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Bakalářskou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím mé bakalářské práce a konzultantem.

Současně čestně prohlašuji, že tištěná verze práce se shoduje s elektronickou verzí, vloženou do IS STAG.

Datum: 14. 5. 2018

Podpis:

Poděkování

Mé poděkování patří Mgr. Janu Březinovi, Ph.D. za vedení mé bakalářské práce, za vstřícnost a cenné rady. Mnoho jsem se díky němu naučil.

Zároveň také děkuji doc. RNDr. Pavlovi Satrapovi, Ph.D. za vytvoření šablony bakalářské práce pro LaTeX, která jistě ušetřila práci mnoha studentům.

Abstrakt

Simulátor proudění podzemní vody s názvem Flow123d (Březina a kol. [2]) využívá za účelem zefektivnění výpočtů s malými vektory a maticemi knihovnu Armadillo, která je schopna tyto výpočty díky optimalizacím například s pomocí výrazových šablon provádět velmi efektivně. Armadillo si však předem pro každou datovou strukturu alokuje prostor v paměti o minimální velikost 16x `double`. To má při použití velkého množství malých vektorů (typicky 3x `double`) negativní dopad na využití cache paměti procesoru a následné zpomalení výpočtů.

V této práci je popsán způsob, jakým lze tuto negativní vlastnost knihovny Armadillo obejít. Byla vytvořena knihovna Armor, která umožňuje provádět výpočty pomocí knihovny Armadillo nad efektivně uloženými daty. Byla provedena analýza využití cache paměti celým programem, následně byla knihovna Armor aplikována ve Flow123d a vliv těchto a dalších optimalizací na běh programu a jeho částí byl otestován. Zároveň byly provedeny celkové testy efektivity využití cache paměti.

Klíčová slova:

Armadillo, C++, cache, Flow123d, vektor, matice, výrazové šablony

Abstract

A simulator of underground water flow named Flow123d (Březina et al. citeFlow123d) uses the Armadillo library for higher efficiency of calculations with small vectors and matrices. This software is capable of performing these calculations very effectively not only thanks to its use of expression templates. Armadillo, however, always prealocates a space in memory of 16 times verb'double'. This has a negative effect on the use of cache when using a large number of small vectors (typically 3 times verb'double') which is then causing a considerable slowdown.

This thesis describes a way to eliminate this negative feature of Armadillo. A new library named Armor was created. This library can perform these calculations using Armadillo with efficiently stored data. An analysis of cache memory usage of the whole program was performed. This library was then applied in Flow123d and the impact of these optimization's on the run-time of the program and its parts was tested. Complete tests of the cache usage efficiency were performed.

Keywords:

Armadillo, C++, cache, expression templates, Flow123d, matrix, vector

Obsah

Seznam tabulek

1 Úvod

1.1 Cíl Práce

Flow123d [2] je výpočetní software využívající k výpočtům s malými maticemi a vektory (zejména o velikosti známé během překladu) knihovnu Armadillo. Původně bylo za tímto účelem uvažováno o několika knihovnách umožňujících efektivní provádění těchto výpočtů. Při porovnávání rychlostí těchto knihoven bylo zjištěno, že knihovna Armadillo, která vykazuje největší výpočetní efektivitu (díky množství optimalizací pomocí tzv. výrazových šablon), trpí jedním nedostatkem. Tímto nedostatkem je neefektivní nakládání s pamětí při použití vektorů a matic s počtem prvků menším než 16x double - typicky např. vektor 3x double (dále jen vektory a matice malých rozměrů). Tento nedostatek a jeho následky jsou dále popsány v kapitole 2.

Za účelem odstranění tohoto nedostatku byl kontaktován autor knihovny. Úpravu ukládání matic s pevnou velikostí nemá autor v plánu. Bylo tedy přikročeno k vlastnímu řešení, které je předmětem této práce.

1.2 Struktura práce

V kapitole 2 je představena knihovna Armadillo a její vlastnosti a nedostatky. V kapitole 3 jsou jednotlivé požadavky a možnosti řešení. V kapitole 4 je představena knihovna Armor jakožto vybrané řešení. V kapitole 5 je provedena analýza využití cache paměti v programu Flow123d a jeho optimalizace s využitím knihovny Armor. Řešení je nakonec prověřeno testy v kapitole 6.

2 Knihovna Armadillo

Tato kapitola se věnuje představení knihovny Armadillo, jejích vlastností, syntaxe a nakonec problematiky této knihovny, jejímž řešením se tato práce zabývá.

Armadillo je C++ knihovna pro výpočty z oblasti lineární algebry. Nabízí syntaxi podobnou Matlabu. Podporuje celá čísla, čísla s plovoucí desetinnou čárkou i komplexní čísla. Pro urychlení vyhodnocení výrazů využívá výrazových šablon a mnoha dalších optimalizací pro zefektivnění výpočtů. Pracuje s vektory a maticemi pevné i proměnné velikosti.

Příklad použití:

```
mat A = randu<mat>(4,5);
mat B = randu<mat>(4,5);
mat C = A * B.t();
```

Původní zdroj upraveného kódu viz [3].

Armadillo si bez ohledu na malý počet prvků vytvářeného vektoru alokuje zbytečně velký prostor v paměti (implicitně 16x double). Tento počet lze změnit pouze globálně.

Využití paměti pro ukládání vektorů tak může být v jistém případě značně neefektivní. Tímto případem jsou již zmíněné vektory a matice malých rozměrů (typicky například vektory o 3 prvcích), přičemž pro výpočty s velkým množstvím takto malých vektorů se při sekvenčním zpracování tato neefektivita negativně projeví na využití cache paměti a i na celkové rychlosti.

Důvodem takovýchto alokací je fakt, že Armadillo bylo původně vytvořeno pro matice a vektory proměnné velikosti. Pokud by chtěl uživatel tedy rozšiřovat počet prvků dané struktury, je pro ony další prvky již před-alokován prostor a rozšíření takové struktury je poté rychlejší. Zároveň je tento způsob ukládání matic a vektorů proměnné velikosti zásadně rychlejší při ukládání do pole a to i navzdory méně efektivnímu využití cache paměti. Možnost použití matic a vektorů o fixní velikosti byla do knihovny přidána později a tato vlastnost již nebyla v knihovně za tímto účelem měněna.

Příčinou proč takovéto nakládání s pamětí činí problém z hlediska efektivity je především využití cache paměti procesoru. Což má za následek v některých případech až několikanásobné zpomalení výpočtů. A to tím způsobem, že jakmile procesoru dojde místo v cache paměti, musí využít přístup do operační paměti. Ta je však mnohonásobně pomalejší a procesor musí vyčkávat na její odpověď. Dle měření vychází, že takovýto výpadek z paměti může procesor zabrzdit řádově na 100 cyklů, pokud jde o čtení, a až dvojnásobně, pokud jde o zápis (cca 200 cyklů). Více o tomto jevu například zde [5].

3 Rozbor požadavků a variant řešení

Problém knihovny Armadillo spočívá v paměťové neefektivitě při použití pro vektory a matice malé velikosti. Řešení musí proto tento problém eliminovat. Cílem je však i zároveň zachovat pozitivní vlastnosti knihovny Armadillo jako je výpočetní efektivita či intuitivní syntaxe.

3.1 Klíčové vlastnosti

Řešení musí být paměťově efektivní pro libovolný počet prvků. Pokud tedy bude třeba například uložit vektor n prvků o velikosti s, měl by tedy v operační paměti alokovat prostor o velikosti $n \cdot s$ či v případě matice $n \cdot m$ prvků o velikosti s prostor o velikosti $n \cdot m \cdot s$.

Knihovna Armadillo je výpočetně velmi efektivní. Využívá celou řadu optimalizací pro urychlení náročných výpočtů například díky využití výrazových šablon. Nejpřínosnějšími optimalizacemi jsou například: odložené vyhodnocování, *loop fusion*, redukce dočasných kopií, vektorizace a další. Více informací o šablonách v C++ například zde [6]. Od hledaného řešení se požaduje tyto optimalizace zachovat.

Jednoduše vysvětlitelným příkladem optimalizace, které Armadillo využívá, může být využití asociativity násobení matic ke snížení počtu prováděných elementárních matematických operací.

Mějme matici A o rozměrech $2 \cdot 3$ (dále jen $A[2,3]$), matici B o rozměrech $3 \cdot 4$ (dále jen $B[3,4]$) a matici C o rozměrech $4 \cdot 1$ (dále jen $C[4,1]$). Při výpočtu součinu těchto matic v následujícím pořadí

$$A[2,3] \times B[3,4] \times C[4,1]$$

lze buď nejprve vyhodnotit součin $A[2,3] \times B[3,4]$ a ten následně vynásobit maticí $C[4,1]$ takto

$$(A[2,3] \times B[3,4]) \times C[4,1]$$

nebo nejprve vyhodnotit součin $B[3,4] \times C[4,1]$ a teprve poté vynásobit matici $A[2,3]$ tímto součinem, tedy následovně.

$$A[2,3] \times (B[3,4] \times C[4,1])$$

V obou těchto případech bude výsledek shodný, co je však klíčové, změní se počet nutných elementárních matematických operací nutných k celkovému vyhodnocení výrazu.

Spočítejme si například počet násobení čísla číslem, ke kterému dojde v jednotlivých případech.

V případě výpočtu $(A[2,3] \times B[3,4]) \times C[4,1]$ se bude provádět $3 \cdot 2 \cdot 4 + 4 \cdot 2 \cdot 1$ tedy celkem 32 operací násobení čísla číslem.

Naopak v případě výpočtu $A[2,3] \times (B[3,4] \times C[4,1])$ se provede $3 \cdot 2 \cdot 1 + 3 \cdot 1 \cdot 4$ tedy 18 operací násobení čísla číslem.

Je tedy zřejmé, že z hlediska rychlosti výpočtu bude výhodné preferovat druhou variantu. A i to jsou optimalizace knihovny Armadillo díky výrazovým šablonám schopny zajistit.

Knihovna Armadillo zároveň nabízí syntaxi podobnou Matlabu (indexace však od nuly). Řešení by proto mělo také nabízet velmi podobnou syntaxi, aby migrace na toto řešení nebyla pro vývojáře softwaru Flow123d nadbytečná zátěž.

Ve fázi nasazení nového řešení do softwaru Flow123d není prakticky možné jednorázově nahradit veškeré využití knihovny Armadillo novým řešením. Nahrazovat je nutné postupně. Nové řešení tedy bude muset zároveň umožňovat koexistenci s knihovnou Armadillo. Knihovny tedy nemohou být vzájemně nekompatibilní. Více o této kompatibilitě v podkapitole 4.1.

3.2 Možnosti řešení

Jednou z možností, která se nabízí, spočívá v úpravě knihovny Armadillo, aby nedocházelo ke zbytečně velkým alokacím paměti pro vektory a matice o malých rozměrech, avšak vzhledem k rozsáhlosti a složitosti této knihovny je toto řešení z časových důvodů nereálné.

Další nabízející se možností je tvorba úplně nové knihovny s vlastní implementací všech funkcí knihovny původní. Toto řešení by však muselo implementovat veškeré výpočetní optimalizace, které nabízí původní knihovna, díky nímž je výpočetně tolik efektivní. Toto řešení by ale bylo natolik časově náročné, že se taktéž jeví zcela nereálné.

Problém s neefektivitou využití cache paměti procesoru nastává jen pokud je neefektivní datová struktura ukládána. Je možné ověřit, zda provádění výpočtů, jak jej definuje Armadillo, nad daty úspornější datové struktury nebude mít za následek v tomto ohledu zefektivnění, což by se projevilo na měřitelných testech rychlostí výpočtů. Toto řešení bylo vybráno k implementaci a je dále rozebráno v kapitole 4.

4 Knihovna Armor

Jedná se o implementaci matice o fixních rozměrech, jejíž ambicí je nahradit implementaci knihovny Armadillo pro její přílišnou paměťovou náročnost (při použití pro malé vektory a matice) a s tím spojenou neefektivitu využití cache procesoru.

Umožňuje vytvářet matice a vektory o zvoleném datovém typu a velikosti a provádět s nimi základní operace jako je sčítání, odčítání, maticové násobení, násobení po složkách, skalární součin, násobení a dělení skalárem a podobně.

Kompletní zdrojový kód knihovny je k dispozici zde [4] v souboru `src/system/armor.hh`.

4.1 Základní princip

Knihovna Armor je koncipována tak, aby zejména snížila paměťovou náročnost implementace knihovny Armadillo.

Ukládá si data do vlastních proměnných, ale k výpočtům využívá implementací jednotlivých operací knihovny Armadillo a to tím způsobem, že jakmile je volán konkrétní operátor, Armor vrátí výsledek operátoru zavolaného na matice či vektory knihovny Armadillo zkonstruovanými nad daty implementace Armor. Metoda pro skalární součin je definována jako obálka odpovídající funkce z knihovny Armadillo.

```
template <class Type, uint nRows, uint nCols>
inline Type dot(const Mat<Type, nRows, nCols> & a,
                const Mat<Type, nRows, nCols> & b) {
    return arma::dot(a.arma(), b.arma());
}
```

Metoda `arma()` vrací vektor knihovny Armadillo. Tato funkce mimo jiné zajišťuje kompatibilitu obou knihoven v rámci softwaru Flow123d. Vždy, kdy je nutné získat Armadillo matici, lze na ni Armor matici jednoduše zkonvertovat. Tato metoda je implementována následujícím způsobem.

```
inline ArmaType arma() const {
    return ArmaType(memptr());
}
```

Metoda `memptr()` vrací ukazatel na první prvek dvourozměrného pole obsahujícího data.

```
inline Type * memptr() {
    return *data;
}
```

Příznivou vlastností tohoto řešení je fakt, že v praxi ke konstrukci Armadillo vektorů zde vůbec nedochází a operace se provede pouze podle jejich předpisu výhradně nad daty původních vektorů. Implementace si tak zachovává veškerou efektivitu výpočtů při zachování nízké paměťové náročnosti.

Ne všechny metody lze však převzít z knihovny Armadillo (například konstruktory, indexace, operátory přiřazení a podobně). Níže vidíme příklad konstruktoru.

```
Mat(std::initializer_list<Type> list) {
    auto it = list.begin();
    for (uint i = 0; i < nRows * nCols; ++i, ++it) {
        *(*data + i) = *it;
    }
}
```

Tento má za úkol vytvořit novou matici a zároveň ji naplnit daty předanými například následujícím způsobem.

```
armor::Mat<2,2> matrix{1, 2, 3, 4};
```

Konstruktor je definován s využitím `for` cyklu namísto např. funkce `std::copy` z důvodu popsaného v sekci 4.4.1.

4.2 Způsob uložení

Data si tato implementace ukládá jako dvourozměrné pole zvoleného datového typu.

Proměnná nesoucí data konkrétně vypadá v kódu následovně.

```
Type data[nCols][nRows];
```

Indexace prvků je v této implementaci, stejně tak jako u matic knihovny Armadillo, vůči klasických dvourozměrným polím transponovaná.

Jednotlivé indexy je pak potřeba na datovou strukturu volat v opačném pořadí.

```
inline Type & operator()(uint row, uint col) {
    return data[col][row];
}
```

Matice je tedy uložena jako vektor sloupcových vektorů.

4.3 Časová náročnost výpočtů

Aby bylo možné ověřit, zda Armor dosahuje cíle v porovnání rychlosti výpočtů s vektory knihovny Armadillo, bylo nutné provést testy. Celkem byly pro tyto účely vytvořeny 3 testy.

- Výpočet sumy velkého množství vektorů - součet přes pole vektorů
- Skalární součin dvou polí vektorů - výpočet skalárního součinu vektorů tak, jak ho implementují jednotlivá řešení
- Násobení matice vektorem - postupné násobení matice různými vektory

Měření se provádělo tak, že se vždy zaznamenal čas před a po provedení výpočtů, časy se odečetly a výsledek převedl na milisekundy. Takto lze určit časovou náročnost výpočtů jednotlivých implementací. Čím je implementace rychlejší, tím kratší je čas výpočtu.

Následuje příklad měření s použitím `std::chrono`, který má rozlišení řádově 10^{-9} sekund.

```
auto start = std::chrono::system_clock::now();

//Výpočet

auto end = std::chrono::system_clock::now();
auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
std::cout << "cas vypoctu: " << elapsed.count() << endl;
```

Hodnoty časů pro jednotlivé testy samy o sobě nehrají roli. Slouží pouze k porovnání jednotlivých implementací.

Výpočty se prováděly vždy opakovaně v cyklu, aby se dosáhlo dostatečného času k zajištění spolehlivosti výsledků.

Armadillo bylo nakonfigurováno, aby nepoužívalo wrapper a aby běželo vždy v režimu vydání nikoli ladění, následujícími příkazy:

```
#define ARMA_DONT_USE_WRAPPER
#define ARMA_NO_DEBUG
```

K zajištění optimalizací při překladu se program překládal pomocí dodatečného argumentu `-O3`.

Testy byly prováděny opakovaně. Výsledný čas je vždy mediánem z výsledků tří testů.

4.3.1 Výpočet sumy vektorů

Při výpočtu sumy je nutné provádět velmi frekventovaně přiřazování, při kterém dochází ke kopírování. Kopírování je jev, který je časově relativně náročný. Tento test spočívá v sečtení velkého množství vektorů do jednoho.

Do tříprvkového vektoru `sum` se postupně přičte 100 000 dalších tříprvkových vektorů. Toto je opakováno 1000× v cyklu pro dosažení dostatečně vypovídajících časů.

```
sum = sum + vector[i];
```

Tabulka 1: Výsledky testu sumy

Implementace	Čas výpočtu (s)
Armadillo	2,72
Armor	0,38

Z výsledků je zřejmé, že implementace Armor zde plní svůj účel a je schopna výpočet urychlit. V tomto případě sedmináso bně.

4.3.2 Výpočet skalárního součinu

Tento test zkoumá rychlost výpočtu skalárního součinu. Jedná se o velmi často využívanou operaci v rámci výpočtů s vektory.

Mějme dvě pole vektorů a třetí pole doublů. Na i-tý prvek třetího pole se ukládá skalární součin i-tého vektoru z prvního a i-tého vektoru z druhého pole.

```
product[i] = dot(vector1[i], vector2[i]);
```

Tabulka 2: Výsledky testu skalárního součinů

Implementace	Čas výpočtu (s)
Armadillo	5,87
Armor	1,03

I v tomto testu je Armor výrazně rychlejší než Armadillo. A to více než pětinásobně.

4.3.3 Výpočet součinu matice a vektoru

V tomto posledním testu je postupně matice násobena různými vektory. Jedná se tedy o výpočetně nejnáročnější test.

```
matArray[i] = mat * vectorArray[i];
```

Tabulka 3: Výsledky testu násobení matice vektorem

Implementace	Čas výpočtu (s)
Armadillo	9,03
Armor	1,18

I v tomto případě výsledky jasně hovoří pro implementaci Armor, která je zde více než sedminásobně rychlejší.

4.4 Faktory ovlivňují rychlost výpočtů

Výsledky testů v podkapitole 4.3 jsou již testovány v ideální konfiguraci vycházející z výsledků testů následujících. Tato podkapitola se věnuje demonstraci vlivů jednotlivých variant těchto konfigurací.

- optimalizace pro automatickou vektorizaci - Při použití dodatečného argumentu `-O3` při překladu může dojít k automatické vektorizaci `for` cyklů v kódu. Což urychlí běh programu.
- změna překladače - Kód přeložený pomocí překladače GCC běží rychleji než kód přeložený pomocí překladače Clang, který však kód rychleji kompiluje.
- změna verze knihovny Armadillo - Novější verze knihovny Armadillo provádí výpočty rychleji než předchozí používaná verze.
- virtualizace Dockeru - Flow123d běží pod Dockerem. I u něj lze očekávat vliv na výkon.

Následující podkapitoly se věnují testování časové náročnosti jednotlivých variant s vlastnostmi, které byly nahrazeny jinými řešeními kvůli jejich negativnímu vlivu na rychlost výpočtů. To jest použití funkcí, u kterých nedochází při překladu k automatické vektorizaci, použití překladače Clang a použití starší verze knihovny Armadillo. V testech je tedy ideální konfigurace vždy porovnána s případem, kdy použijeme u daného faktoru pomalejší variantu se zachováním rychlejších variant ostatních faktorů. I zde byly testy prováděny opakovaně. Jako výsledný čas byl pro každý test zvolen medián ze tří naměřených hodnot.

4.4.1 Vektorizace

Bylo zjištěno, že při použití dodatečného argumentu `-O3` při překladu programu může dojít k automatické vektorizaci `for` cyklů. Pokud například takto překopírováváme hodnoty z jedné matice do druhé, mohou se všechny tyto hodnoty překopírovat současně jedinou instrukcí procesoru.

Za tímto účelem je tedy vhodné vždy v těchto případech použít `for` cyklus, který umí překladač vektorizovat namísto využití jiného řešení.

Právě ono zmíněné překopírování hodnot lze buď provést například s využitím fukce `std::copy` ze standardní knihovny C++, přičemž nedojde k automatické vektorizaci nebo právě s využitím klasického `for` cyklu, kde je již během překladu zřejmé, kolik hodnot se bude kopírovat. To je klíčová vlastnost nutná pro automatickou vektorizaci.

```
inline const Mat<Type, nRows, nCols> & operator=(const ArmaType & other) {
    std::copy(other.begin(), other.end(), begin());
    return *this;
}
```

```
inline const Mat<Type, nRows, nCols> & operator=(const ArmaType & other) {
    for (uint i = 0; i < nRows * nCols; ++i) {
        data[0][i] = other[i];
    }
    return *this;
}
```

Výsledkem pak může být výrazné urychlení některých výpočtů, pokud k automatické vektorizaci dojde. Tento jev lze nejlépe pozorovat například u výpočtu sumy vektorů, kde právě překopírování prvků tvoří výraznou část činnosti procesoru při jeho provádění.

Po provedení testů zde skutečně je znatelný rozdíl v rychlosti.

Tabulka 4: Test vlivu vektorizace

Implementace	Čas výpočtu (s)
`std::copy`	0,79
`for` cyklus	0,38

Vektorizace je v tomto případě zodpovědná za zdvojnásobení rychlosti výpočtu. V implementaci Armor jsou proto pro takto vektorizovatelné oparace použity `for` cykly.

4.4.2 Překladač

Dále bylo zjištěno, že na rychlost výpočtů má významný vliv překladač, kterým je kód kompilován. Byly uvažovány dva překladače. Clang a GCC. Ačkoli GCC překládá kód výrazně pomaleji než Clang, kód zkompilovaný tímto překladačem běží rychleji, což je klíčové.

Překladače byly porovnány ve všech třech výpočtech s použitím implementace Armor. V testu skalárního součinu byla rychlost v obou případech prakticky totožná, v testu sumy vektorů a násobení matice vektorem však vykazuje GCC výrazně lepší výsledky.

Tabulka 5: Test vlivu překladače - suma vektorů

Překladač	Čas výpočtu (s)
Clang	2,99
GCC	0,38

Tabulka 6: Test vlivu překladače - násobení matice vektorem

Překladač	Čas výpočtu (s)
Clang	4,20
GCC	1,18

Z extrémního rozdílu v rychlosti v testu sumy vektorů lze usoudit, že překladač Clang mimo dalších vlivů pravděpodobně neprovádí automatickou vektorizaci. Každopádně výsledky hovoří ve prospěch GCC. Z tohoto důvodu je pro kompilaci kódu použit právě tento překladač.

4.4.3 Verze knihovny Armadillo

Dalším faktorem ovlivňujícím rychlost výpočtů je použitá verze knihovny Armadillo. Byly provedeny testy porovnávající rychlost ve všech třech výpočtech při použití starší verze knihovny s novější verzí 7.800. Změny rychlosti jsou patrné pouze v testu sumy vektorů a testu násobení matice vektorem. V testu skalárního součinu je rychlost prakticky bez efektu.

Tabulka 7: Test vlivu verze knihovny Armadillo - suma vektorů

Verze knihovny	Čas výpočtu (s)
Původní verze	0,33
Verze 7.800	0,38

Tabulka 8: Test vlivu verze knihovny Armadillo - násobení matice vektorem

Verze knihovny	Čas výpočtu (s)
Původní verze	3,68
Verze 7.800	1,18

Lze si všimnout, že novější verze knihovny způsobila při výpočtu sumy mírné zpomalení (konkrétně 15 %), avšak v případě násobení matice vektorem pozorujeme trojnásobné zrychlení.

Z tohoto důvodu je výsledně použita novější verze knihovny.

4.4.4 Docker

Docker je nástroj poskytující virtuální prostředí, ve kterém je například možné spustit program bez ohledu na typ operačního systému. Více o dockeru zde [7].

Výsledky testů vlivu Dockeru jsou však překvapivé. Očekávali bychom zpomalení z důvodu dodatečné režie nutné k virtualizaci, ale výsledky vykazují nepatrné zrychlení V testu sumy vektorů činí zrychlení přes 9,5 %.

Tento jev se projevil i při zvýšení počtu opakování měření ze 3 na 15. Operační systém, na kterém byly časy naměřeny byl KDE Neon LTS User Edition 5.12.

Tabulka 9: Test vlivu verze knihovny Armadillo - násobení matice vektorem

Verze knihovny	Čas výpočtu (s)
Systém	0,42
Docker	0,38

5 Použití knihovny Armor ve Flow123d

Po ověření správnosti řešení pomocí testů při dílčích elementárních výpočtech přichází na řadu nasazení tohoto řešení ve Flow123d. Nejprve je třeba nalézt klíčová místa v kódu, kde je vhodné Armor použít.

5.1 Analýza pomocí nástroje Cachegrind

Nástroj Cachegrind je software umožňující spustit program ve virtuálním prostředí a monitorovat při tom jeho nakládání s pamětí včetně výpadků z jednotlivých úrovní cache paměti procesoru. Díky tomu je možné zmapovat klíčová místa ve Flow123d, kde dochází k největším výpadkům z cache paměti.

Cachegrind je takto schopen poskytnout informace o jednotlivých funkcích programu. Například počet čtení, zápisů, výpadků při čtení a výpadků při zápisech pro konkrétní úroveň cache paměti. Zároveň umožňuje výstup seřadit dle zvolené veličiny. Více o tomto nástroji zde [1].

Pro detailnější informace byl vytvořen jednoduchý skript v jazyce Python pro rozšíření výstupu o poměr počtu výpadků při čtení ku počtu čtení, poměr počtu výpadků při zápisu ku počtu zápisů a konečně kalibrovaná hodnota součtu dvojnásobku druhého poměru a prvního poměru, jelikož výpadek při zápisu má přibližně dvakrát větší dopad na čas, po který je procesor nečinný při čekání na odpověď operační paměti. To vše pouze pro výpadky z L2 cache neboť právě ta má nejvýraznější vliv na čas běhu programu. Takováto hodnota (dále jen kritérium) tedy vypovídá o míře neefektivity dané funkce. Skript zároveň automaticky seřadí výstup dle této hodnoty.

Tabulka 10: Zjednodušený výtah z výstupu z nástroje Cachegrind

index	kritérium	Dr	Dw	file:function
1	0.7070297057	5,442,424	2,722,564	`???:VecWAXPY_Seq`
2	0.3954962738	574,139	12,260,731	`memset.S:__GI_memset`
3	0.3187621246	129,006,454	5,850,272	`???:MatMult_SeqAIJ`
4	0.2499992847	0	2,097,166	`simple_allocator.hh: create_buckets(ulong)`

Nejzásadnější zjištění, ke kterému během této analýzy došlo, naznačuje, že daleko vyšší vliv na rychlost asemblace má jiný faktor, který nespadá do zadání této

práce. Výsledný naměřený efekt nahrazení knihovny Armadillo knihovnou Armor tedy nebude mít ve výsledku tak značný vliv na rychlost asemblace, jak se původně předpokládalo.

Pro paralelní zpracování matic velkých rozměrů Flow123d používá knihovnu PETSc. Více o této knihovně např. zde [8]. První a třetí uvedená funkce v tabulce jsou funkce přímo této knihovny. Čtvrtá funkce v tabulce je funkce, kterou volá profiler Flow123d. Druhá funkce v tabulce je obecná systémová funkce, u které bylo třeba zjistit, odkud je volána s pomocí nástroje Callgrind (viz [9]), díky kterému bylo zjištěno, že za téměř všechna tato volání je opět zodpovědná knihovna PETSc. Příčinou tohoto nehospodárného chování metod této knihovny je pravděpodobně neoptimální uspořádání řádků a sloupců matic, se kterými knihovna pracuje. Toto je předmětem dalšího zkoumání.

Dále již nebylo postupováno na základě výsledků z Cachegrindu, ale na základě odhadu vhodných míst v kódu a bylo nutné provést měření dílčích funkcí programu.

5.2 Klíčová místa pro aplikaci nového řešení

Výčet nejvýznamnějších míst v kódu, kde byla použita knihovna Armor ke zrychlení běhu programu.

5.2.1 Point

V souboru `src/mesh/point.hh` se nachází definice Třídy `Space`. Uvnitř této třídy se nachází definice vektoru o zvolené délce a byl sem přidán vektor knihovny Armor pojmenován jako `APoint`.

```
typedef typename armor::vec<spacedim> APoint;
```

5.2.2 Bih tree

Z pohledu výkonu nejzásadnější změnou je použití Armor vektoru v souboru `src/mesh/bih_tree.cc`, kde je přidána definice metody `find_point`, která využívá alias `Space<3>::APoint`.

5.2.3 Bounding box

Třída `BoundingBox` si pak v souboru `src/mesh/bounding_box.hh` definuje alias `Point` jako `Space<dimension>::APoint`. Tuto datovou strukturu pak využívá jako své krajní vrcholy.

5.2.4 Node

Dalším z nejdůležitějších míst, na kterých byl vektor knihovny Armadillo nahrazen vektorem knihovny Armor byl soubor `src/mesh/nodes.hh`. Třída `Node` zde využívá třírozměrný vektor k uložení souřadnic (bod v třírozměrném prostoru).

```
armor::vec<3> coordinates;
```

Metoda vracející tento vektor souřadnic byla pro Armor implementována následovně:

```
inline const armor::vec<3> & point_armor() const {
    return coordinates;
}
```

V zájmu zachování zpětné kompatibility bylo nutné ponechat metodu, která by byla schopna vracet stále třírozměrný vektor knihovny Armadillo konverzí metodou `arma()` (implementace v podkapitole 4.1).

```
inline arma::vec3 point() const {
    return coordinates.arma();
}
```

5.2.5 Element

V neposlední řadě byla přidána optimalizace, která zdánlivě nesouvisí s nahrazením knihovny. Jedná se o přidání metody pro před-výpočet středu.

```
void Element::precalculate_centre() {
    unsigned int li;

    central_point.zeros();

    FOR_ELEMENT_NODES(this, li) {
        central_point += node[ li ]->point_armor();
    }
    central_point /= (double) n_nodes();
}
```

Na první pohled by se mohlo zdát, že připravovat si před-výpočet je v každém případě efektivnější než počítat střed vždy znovu, když je potřeba, aniž bychom si výsledek ukládali. Při použití knihovny Armor byl negativní dopad uložení v daném kontextu tolik paměťově neefektivní datové struktury naopak výraznější než samotné opakování výpočtu. Toto knihovna Armor pomohla změnit, čímž umožnila implementaci této optimalizace a následné zrychlení běhu programu.

6 Ověření testy

Nakonec je třeba tyto změny prověřit testy rychlosti. Kromě měření času běhu asemblace, který vyžaduje přímo zadání, byl vytvořen další test měřící rychlost běhu té části programu, ve které se nejintenzivněji počítá právě s vektory malé velikosti. Právě na tuto oblast nedostatků knihovny Armadillo knihovna Armor cílí.

Testy byly prováděny pomocí vlastního testovacího profileru, který Flow123d nabízí. Výsledky měření profiler zapíše do JSON souborů, které jsou následně zpracovány jednoduchým Python skriptem, který vypočte medián a vypíše výsledky pro jednotlivé varianty. Skripty jsou k dispozici na přiloženém CD.

6.1 Bih tree test

Za účelem otestování rychlosti Bih tree byl vytvořen nový test, který tak testuje tu část programu, ve které dochází nejvíce k výpočtům s velkým množstvím malých vektorů.

Vzhledem k tomu, že se jedná o unit test, je ve Flow123d spouštěn v Dockeru příkazem `make bih_armor-1-test` spuštěným ve složce `src/unit_tests/mesh`, kde se nachází jeho zdrojový kód v souboru `bih_armor_test.cpp`.

Test provádí tři měření. `find_point`, `find_bounding_box` a `create_tree`.

Rychlost běhu těchto metod je ovlivněna změnami

Tabulka 11: Výsledky testu find point

Implementace	Čas výpočtu (s)
Armadillo	0,23
Armor	0,17

Tabulka 12: Výsledky testu find bounding box

Implementace	Čas výpočtu (s)
Armadillo	3,07
Armor	2,15

Tabulka 13: Výsledky testu create tree

Implementace	Čas výpočtu (s)
Armadillo	8,08
Armor	1,47

Celkově tedy z testů vyplývá, že při použití knihovny Armor je doba těchto výpočtů třetinová.

6.2 Test asemblace

Na závěr je třeba otestovat rychlost asemblace. Test lze spustit ze složky `tests/14_darcy_richards` příkazem: `../../bin/flow123d 02_1d_dirichlet.yaml`. Výsledné JSON soubory s naměřenými hodnotami se při tom uloží do složky `tests/output`.

Tabulka 14: Výsledky testu create tree

Implementace	Čas výpočtu (s)
Armadillo	3,72
Armor	2,55

Je zde patrné zrychlení o necelých 46 %. Jak již bylo řečeno v podkapitole 5.1, během analýzy programu pomocí nástroje Cachegrind bylo zjištěno, že na rychlost asemblace má výraznější vliv jiný faktor. Zrychlení proto není tak masivní, jak se původně očekávalo.

7 Závěr

V rámci této práce jsem se obeznámil s výpočetním softwarem Flow123d včetně jeho závislosti na knihovně Armadillo, kterou Flow123d využívá pro výpočty s maticemi a vektory. Obeznámil jsem se s principem a využitím výrazových šablon, které knihovna Armadillo používá k zefektivnění výpočtů. V rámci plnění cíle práce, který spočíval v odstranění nedostatku knihovny Armadillo (neefektivnímu využití cache paměti procesoru pro vektory a matice malých rozměrů) jsem využil v rámci vlastní knihovny nahrazující knihovnu Armadillo paměťové optimalizace prostřednictvím vytvoření vlastní efektivní datové struktury včetně využití stávajících optimalizací pomocí výrazových šablon původní knihovny.

Pomocí nástroje Cachegrind jsem monitoroval využití cache paměti procesoru algoritmů provádějících asemblaci matic. Otestoval jsem a aplikoval na klíčových místech optimalizace v datových strukturách.

Ačkoli bylo během analýzy pomocí nástroje Cachegrind odhaleno, že nejzásadnější vliv na snížení rychlost asemblace mají funkce knihovny PETSc, která nespadá do zadání této práce, a pro ověření správnosti je tudíž vhodnější Bih tree test, ve kterém naměřené zrychlení činí přibližně 200 %, zadání udává demonstrovat výsledky na testu asemblace, což se i přes výrazně menší vliv než byl očekáván přesto podařilo. Následkem je, že dopad této optimalizace zde není tak zásadní, jak se očekávalo, avšak zrychlení činí necelých 46 %. Práce zároveň tímto přispěla k odhalení vlivnějšího faktoru a na základě tohoto odhalení se podnikají další kroky k zefektivnění tohoto softwaru.

Podařilo se tedy splnit zadání a přispět tak k vylepšení Flow123d eliminováním zásadního nedostatku jedné z jeho knihoven. A podařila se tak prokázat správnost strategie řešení, kterou knihovna Armor představuje. Kompletní zdrojové kódy upraveného softwaru Flow123d jsou k dispozici zde [4].

Osobně mi tato práce pomohla ujistit se v mém budoucím směřování. Velmi výrazně jsem se během práce na ni zdokonalil v programování v jazyce C++, nabral jsem nové zkušenosti s výpočetním softwarem a naučil se psát optimalizovaný kód pro maximální efektivitu z hlediska rychlosti běhu programu a efektivního využití hardwarových prostředků.

Během zkoumání vlivů na rychlost výpočtů došlo k překvapivému zjištění, že test běží rychleji pod Dockerem než mimo něj. Očekáváno bylo opačné chování, kdy virtualizace má vliv na zpomalení kvůli nutné režii. Toto však nenastalo a v rámci této práce nebyl tento jev vysvětlen, neboť důvodem měření byla snaha o zjištění nutnosti spouštět program mimo Docker v případě výrazného opačného výsledku testu. Toto se neprokázalo, takže nebylo nutné se tím dále zabývat.

Mezi klíčové přínosy této práce patří prokázání zrychlení pomocí knihovny Armor v testech, zrychlení Bih tree, dílčí urychlení asemblace a širší analýza využití cache paměti ve Flow123d.

Práce zároveň pomohla odhalit nutnost optimalizovat uspořádání velkých vektorů a matic při použití knihovny PETSc, což je vhodné téma diplomové práce.

Literatura

[1] Cachegrind: a cache and branch-prediction profiler. *Valgrind* [online]. [cit. 2018-05-14]. Dostupné z: *http://valgrind.org/docs/manual/cg-manual.html*

[2] Flow123d. *Flow123d* [online]. [cit. 2018-05-14]. Dostupné z: *http://flow123d.github.io/*

[3] API Documentation for Armadillo 8.500. *Armadillo: C++ library for linear algebra & scientific computing* [online]. [cit. 2018-05-14]. Dostupné z: *http://arma.sourceforge.net/docs.html*

[4] GitHub. *Flow123d/flow123d at PK_bih_tree_calibrated* [online]. [cit. 2018-05-14]. Dostupné z: *https://github.com/flow123d/flow123d/tree/PK_bih_tree_calibrated*

[5] LEVINTHAL, David. *Performance Analysis Guide for Intel. R CoreTM i7 Processor and Intel R XeonTM, 5500.*

[6] VANDEVOORDE, David. a Nicolai M. JOSUTTIS. *C++ templates: the complete guide.* Boston, MA: Addison-Wesley, 2003. ISBN 02-017-3484-2.

[7] What is Docker. *What is Docker?* [online]. [cit. 2018-05-14]. Dostupné z: *https://www.docker.com/what-docker*

[8] Documentation. *PETSc: Documentation* [online]. [cit. 2018-05-14]. Dostupné z: *https://www.mcs.anl.gov/petsc/documentation/index.html*

[9] Callgrind: a call-graph generating cache and branch prediction profiler. *Valgrind* [online]. [cit. 2018-05-16]. Dostupné z: *http://valgrind.org/docs/manual/cl-manual.html*

Obsah přiloženého CD

- Text bakalářské práce v elektronické podobě
- Data k výsledkům z prověřovacích výpočtů Armoru
- Data k výsledkům z testu Bih tree
- Data k výsledkům z testu asemblace
- Výstupní data z nástroje Cachegrind
- Zdrojový kód knihovny Armor